Omni-directional Shadows

Peter Houska

Institute of Computer Graphics and Algorithms

Vienna University of Technology

Omni-directional Shadows

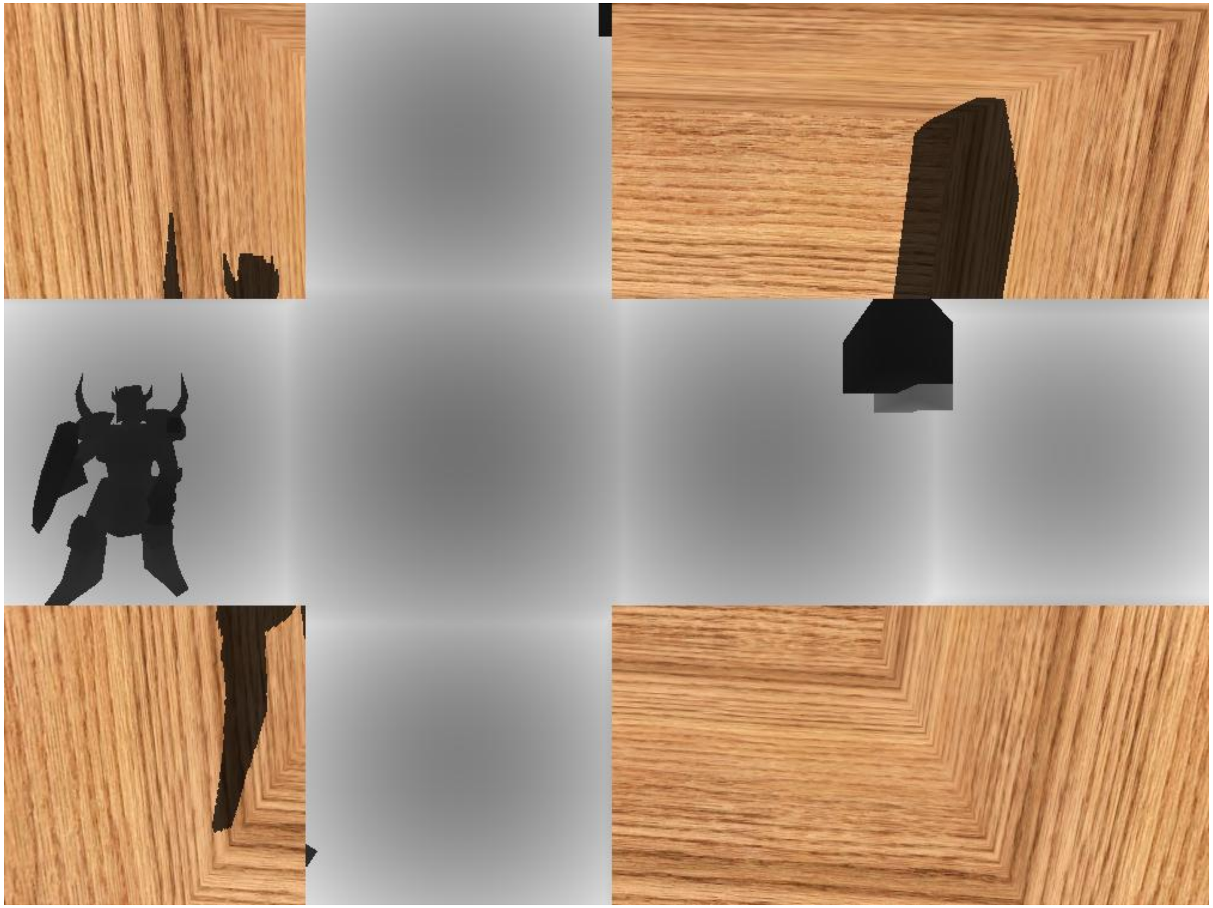
- In reality we often encounter light sources which cast light “in all directions”
  - Lightbulbs everywhere ;-)
- It follows that shadows are cast in all directions
  - We want to capture this effect in realtime applications, too!

Omni-directional Shadows

- Two common techniques
  - Omni-directional Shadow Maps
  - Shadow Volumes
- Modern GPUs and APIs expose extremely useful functionality
  - Especially Geometry Shader alleviates many tasks involved
- Omni-directional Shadows nowadays both fast and easy to implement

Overview

- **Omni-directional Shadow Maps**
- Shadow Volumes



Traditional Shadow Maps

- Established technique
  - Lance Williams, “Casting Curved Shadows on Curved Surfaces”, 1978
- Shadow Mapping works perfectly for camera-like light sources
  - Directional light
  - Spotlight
- What about point lights?
  - Should be casting shadows “in all directions”

Enhancing Shadow Mapping - Method 1

- Use traditional “light source camera“
  - Must have 90° FOV
- Orient „light source-camera“ along main world space axes (+x,-x,+y,-y,+z,-z)
- Render each direction individually and write depth to 6 separate textures
- Obviously 6 render passes and 6 shadow maps needed
- No additional GPU features needed

Enhancing Shadow Mapping - Method 2

- Use traditional “light source camera“
  - Must have 90° FOV
- Orient „light source-camera“ along main world space axes (+x,-x,+y,-y,+z,-z)
- Render each direction individually and write depth to 1 cubemap texture in 1 pass
- Geometry shader can
  - Duplicate rendered geometry
  - Transform according to each viewing direction
  - Dispatch fragments to proper cubemap face

Creating a Geometry Shader Object

- Completely analogous to creating vertex and fragment shader objects
- Only difference
  - glCreateShader(GL_GEOMETRY_SHADER);
  - … instead of GL_VERTEX_SHADER / GL_FRAGMENT_SHADER
- Additionally to vertex- and fragment shader objects, attach geometry shader object to program object
  - glAttachShader(program_obj, shader_obj);

Creating the Depth Cubemap FBO

- Create depth-cubemap texture
  - Consists of six 2D depth textures
  - One for each face with target set to:

```
GL_TEXTURE_CUBE_MAP_POSITIVE_X
GL_TEXTURE_CUBE_MAP_NEGATIVE_X
GL_TEXTURE_CUBE_MAP_POSITIVE_Y
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
GL_TEXTURE_CUBE_MAP_POSITIVE_Z
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
//or equivalently: GL_TEXTURE_CUBE_MAP_POSITIVE_X+i, i=0..5
```

- Create FBO
- Attach cubemap texture at FBO‘s depth attachment point

Creating the Depth Cubemap Texture

```
// depth cubemap texture
GLint texID;
glGenTextures(1, texID);
glBindTexture(GL_TEXTURE_CUBE_MAP, texID);


// fixes seam-artifacts due to numerical precision limitations
glTexParameteri(      GL_TEXTURE_CUBE_MAP,
                      GL_TEXTURE_WRAP_S,
                      GL_CLAMP_TO_EDGE      );
// equivalent calls for
// GL_TEXTURE_WRAP_T and GL_TEXTURE_WRAP_R, respectively


glTexParameteri(      GL_TEXTURE_CUBE_MAP,
                      GL_TEXTURE_MAG_FILTER,
                      GL_LINEAR             );
// equivalent call for GL_TEXTURE_MIN_FILTER
```

Creating the Depth Cubemap Texture

```
// traditional 24 bit unsigned int z-buffer
GLint internal_format =  GL_DEPTH_COMPONENT24;
GLenum data_type = GL_UNSIGNED_INT;

// float z-buffer (if more precision is needed)
// GLint internal_format =  GL_DEPTH_COMPONENT32F;
// GLenum data_type = GL_FLOAT;

GLenum format = GL_DEPTH_COMPONENT;

for (GLint face = 0; face < 6; face++) {
        glTexImage2D(  GL_TEXTURE_CUBE_MAP_POSITIVE_X + face,
                       0,
                       internal_format,
                       texW, texH, 0,
                       format,
                       data_type,
                       NULL //content need not be specified
                       );
}
```

Attaching Cubemap to FBO

```
//create FBO
GLuint texFBO;
glGenFramebuffers(1, texFBO);
glBindFramebuffer(GL_FRAMEBUFFER, texFBO);

//attach depth cubemap texture to FBO’s depth attachment point
glFramebufferTexture( GL_FRAMEBUFFER,
                      GL_DEPTH_ATTACHMENT,
                      texID, 0              );

//Tell OpenGL that we are aware of the fact, that we did not
//attach a color texture. If we didn‘t do this, OpenGL would
//consider the FBO as incomplete.
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);

// later, when wishing to render to FBO:
glBindFramebuffer(GL_FRAMEBUFFER, texFBO);
glViewport(0, 0, texW, texH);
```

Orienting and Positioning the Cameras

- Each of the 6 cameras must be placed into the scene correctly
  - Calculate their view matrices
  - Split into rotational part …
    - Unique for each camera
  - … and translational part
    - The same for all cameras

The 6 View Matrices (Rotations)

$$pos\_x = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad neg\_x = \begin{bmatrix} 0 & 0 & +1 & 0 \\ 0 & -1 & 0 & 0 \\ +1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$pos\_y = \begin{bmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad neg\_y = \begin{bmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$pos\_z = \begin{bmatrix} +1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad neg\_z = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 6 View Matrices (Translations)

- Calculate a translation matrix that translates by –light_pos

The 6 View Matrices

- Combine the matrices
  - $V[i] = R[i]\,T\ ,\ i \in [0; 5]$
    - V[i] … view matrix for camera i
    - R[i] … rotational part of view matrix i
    - T … translational part

1st Pass – Render to Cubemap

- Bind Depth Cubemap FBO
  - Don‘t forget to call `glClear(GL_DEPTH_BUFFER_BIT)`
- Calculate 6 view matrices V[6]
- Pass P*V[i] to shader, where
  - P … 90° FOV projection matrix
  - Keeping near- and far plane close together can help improve depth precision
- Render geometry
  - no textures, lighting, …

R2CM Vertex Shader

```
#version 330 core

uniform mat4 M_mat; //model matrix (passed per object)

in vec3 attr_vertex; // object space vertex positions

void main(void) {

       // transform vertex to world space
       gl_Position = M_mat * vec4(attr_vertex, 1.0);

       // in the GS the value of gl_Position can
       // be accessed like this:
       // gl_in[“triangle_vertex_idx“].gl_Position;
}
```

R2CM Geometry Shader

```
#version 330 core
layout(triangles) in;
//3 vertices per tri, output 6 tris (1 for each cm-face)
layout(triangle_strip, max_vertices=18) out;

// contains P*V[i], transforms from WS to cubemap-face i
uniform mat4 cm_mat[6];

out vec4 WS_pos_from_GS;

void main(void) {
  //iterate over the 6 cubemap faces
  for(gl_Layer=0; gl_Layer<6; ++gl_Layer) {
    for(int tri_vert=0; tri_vert<3; ++tri_vert) {
       WS_pos_from_GS = gl_in[tri_vert].gl_Position;
       gl_Position = cm_mat[gl_Layer] * WS_pos_from_GS;
       EmitVertex();
    }
    EndPrimitive();
  }
}
```

R2CM Geometry Shader

- gl_Layer special built-in variable
  - Redirects fragments to different cubemap faces

Layer Number	Cubemap Face
0	GL_TEXTURE_CUBE_MAP_POSITIVE_X
1	GL_TEXTURE_CUBE_MAP_NEGATIVE_X
2	GL_TEXTURE_CUBE_MAP_POSITIVE_Y
3	GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
4	GL_TEXTURE_CUBE_MAP_POSITIVE_Z
5	GL_TEXTURE_CUBE_MAP_NEGATIVE_Z

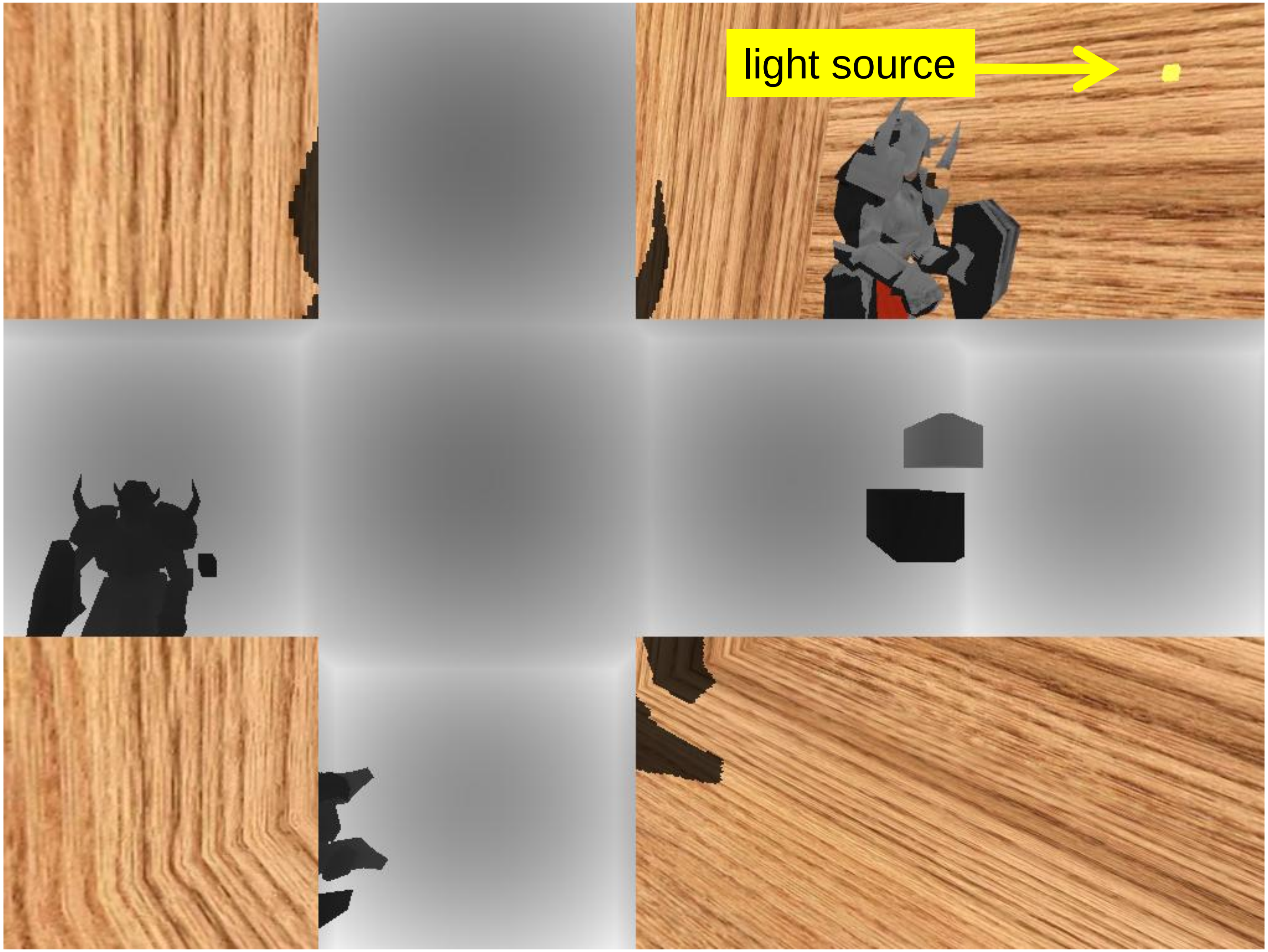
R2CM Fragment Shader

```
#version 330 core

uniform vec2 near_far; // near and far plane for cm-cams
uniform vec4 l_pos; // world space light position

in vec4 WS_pos_from_GS;

void main(void) {

  // calculate distance
  float WS_dist = distance(WS_pos_from_GS, l_pos);

  // map value to [0;1] by dividing by far plane distance
  float WS_dist_normalized = WS_dist / near_far.y;

  // write modified depth
  gl_FragDepth = WS_dist_normalized;

  // when using depth-only FBO, do NOT write to color!!!
}
```

1^{st} Pass – Render to Cubemap

- Now cubemap stores the distances to the objects which the light rays would hit first
  - Just as in traditional shadow mapping

Depth Values Stored in CM


light source

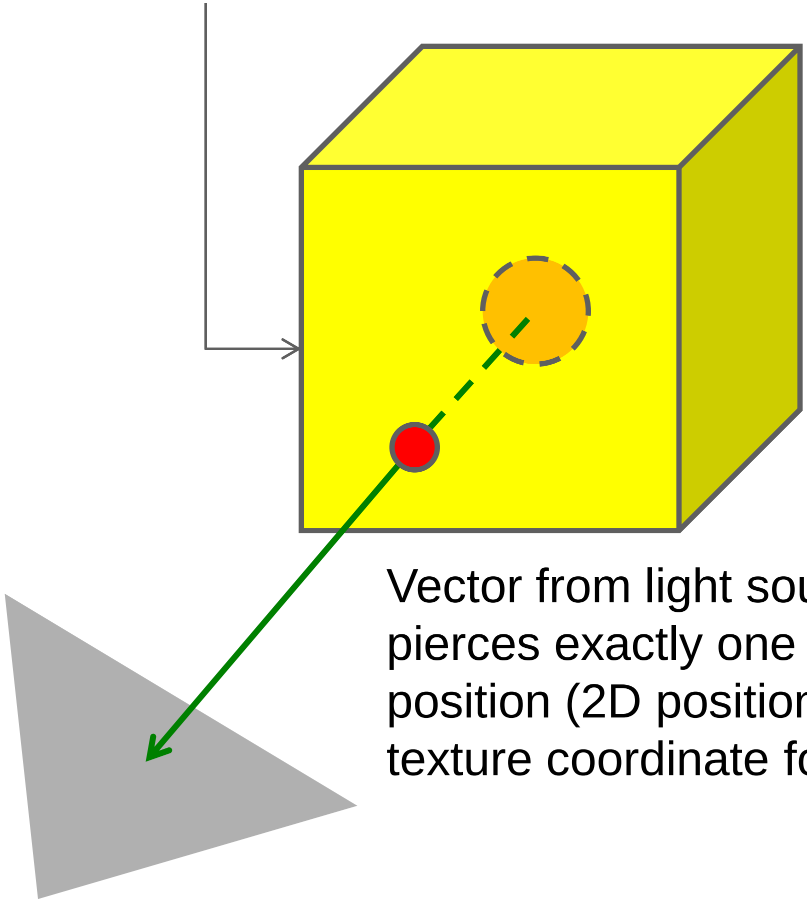
2nd Pass

- Render lit scene
- Use information from 1st pass to determine shadowed regions
  - Same basic idea as in traditional SM, but different lookup needed for cubemap
- Cubemap “situated“ in world space
- Depth values stored are scaled distances from object to light source

2nd Pass

- Use vector from surface position to light position
  - Vector has direction and magnitude
  - Direction is used as the texture coordinate to address the cubemap
    - Now we have smallest distance d_l from light to scene
  - Magnitude gives us distance d_s of current surface point to light source
  - Compare these distances
    - If d_l < d_s the surface point lies in shadow

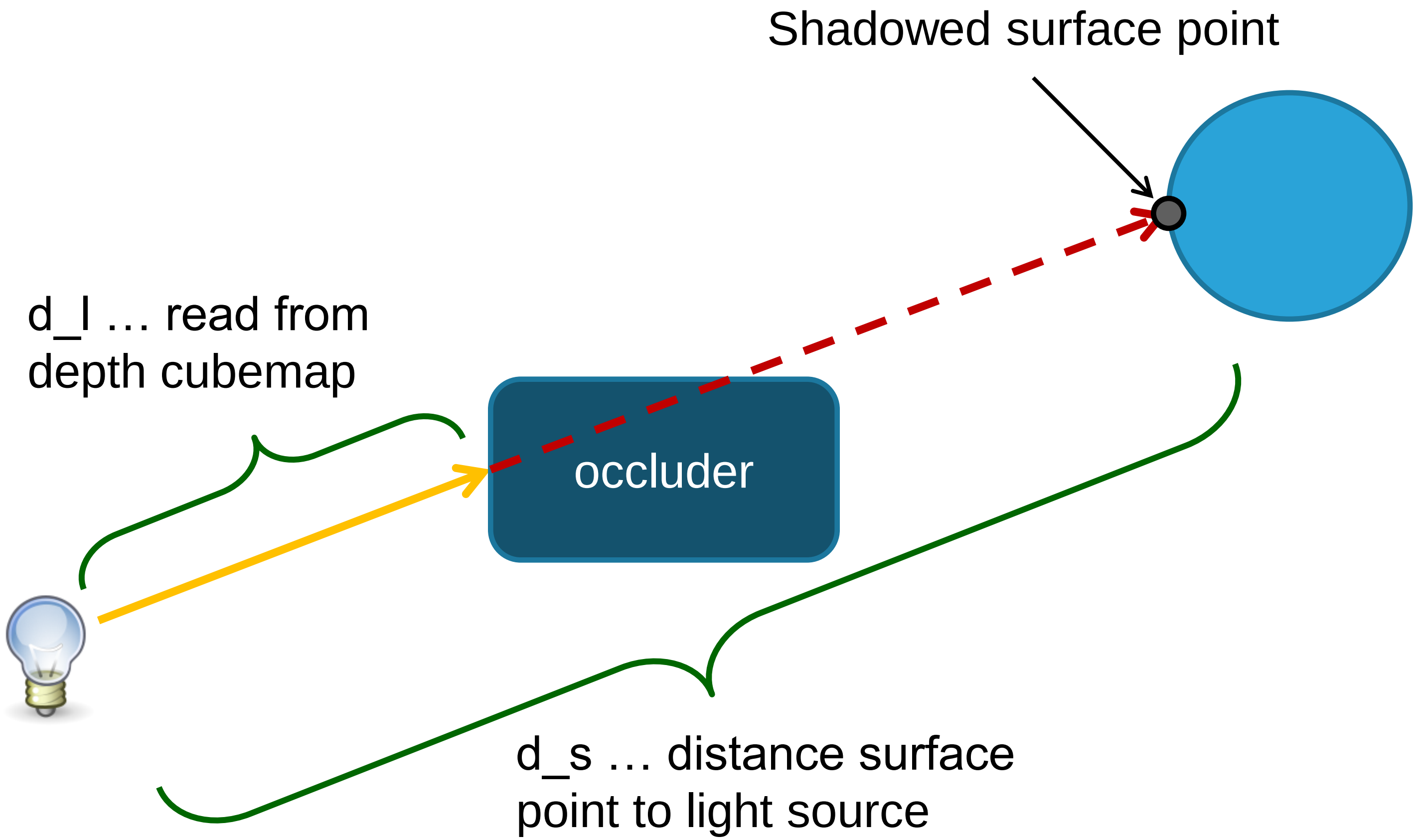
2nd Pass – Cubemap Texture Coords



cubemap centered around light source

Vector from light source to scene-surface point pierces exactly one cubemap face at a specific position (2D position on this face) ➔
texture coordinate for cubemap lookup!

2nd Pass – Comparing Distances



Shadowed surface point

d_l … read from
depth cubemap

occluder

d_s … distance surface
point to light source

2nd pass: Vertex Shader

```
#version 330 core

in vec3 attr_vertex; // vertex position

uniform mat4 M_mat; //model matrix (passed per object)

// ...
// additional attributes (vertex normal, tex-coord, ...)
// and uniforms (other matrices etc.)
// ...

out vec4 WS_pos;

void main(void) {

  // ...
  WS_pos = M_mat * vec4(attr_vertex, 1.0);
  // ...
}
```

2nd pass: Fragment Shader (1/2)

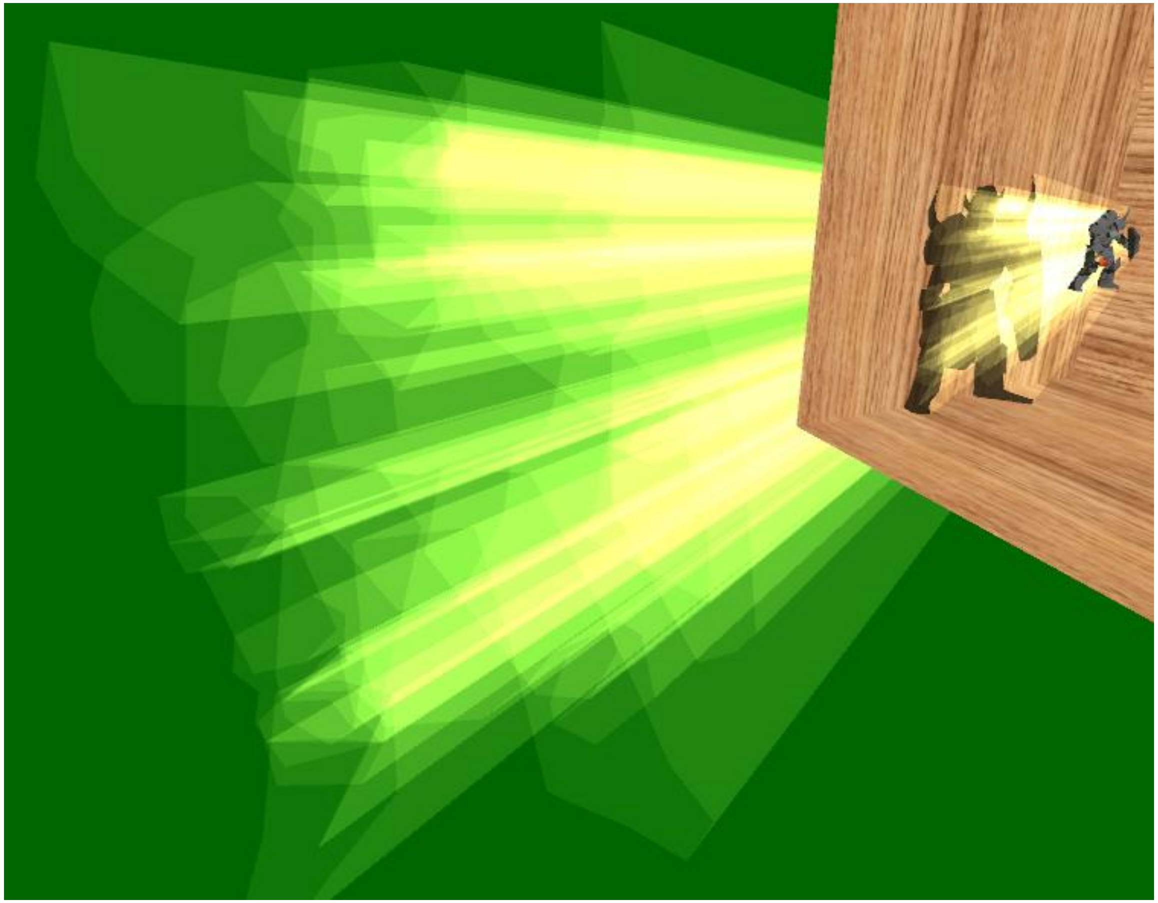
```
#version 330 core

uniform samplerCube cm_z_tex;

uniform vec4 l_pos; //world space light position

uniform vec2 near_far; // near and far plane for cm-cams

in vec4 WS_pos;

// ... Possibly other uniforms and varyings
```

2nd pass: Fragment Shader (2/2)

```
void main(void) {
  // calculate vector from surface point to light position
  // (both positions are given in world space)
  vec3 cm_lookup_vec = WS_pos.xyz - l_pos.xyz;

  // read depth value from cubemap shadow map
  float smallest_dist_to_light = texture(cm_z_tex, cm_lookup_vec).r;

  // WS “dist-to-lightsource” for current fragment
  float curr_fragment_dist_to_light = length(cm_lookup_vec);

  // undo previous [0;1]-mapping of ”dist-to-lightsource”
  smallest_dist_to_light *= near_far.y;

  float eps = 0.15; // add a small offset (adjust as needed)
  if(smallest_dist_to_light+eps < curr_fragment_dist_to_light)
       // ==> fragment lies in shadow

  // perform other calculations, then set fragment’s color
}
```

Overview

- Omni-directional Shadow Maps
- **Shadow Volumes**

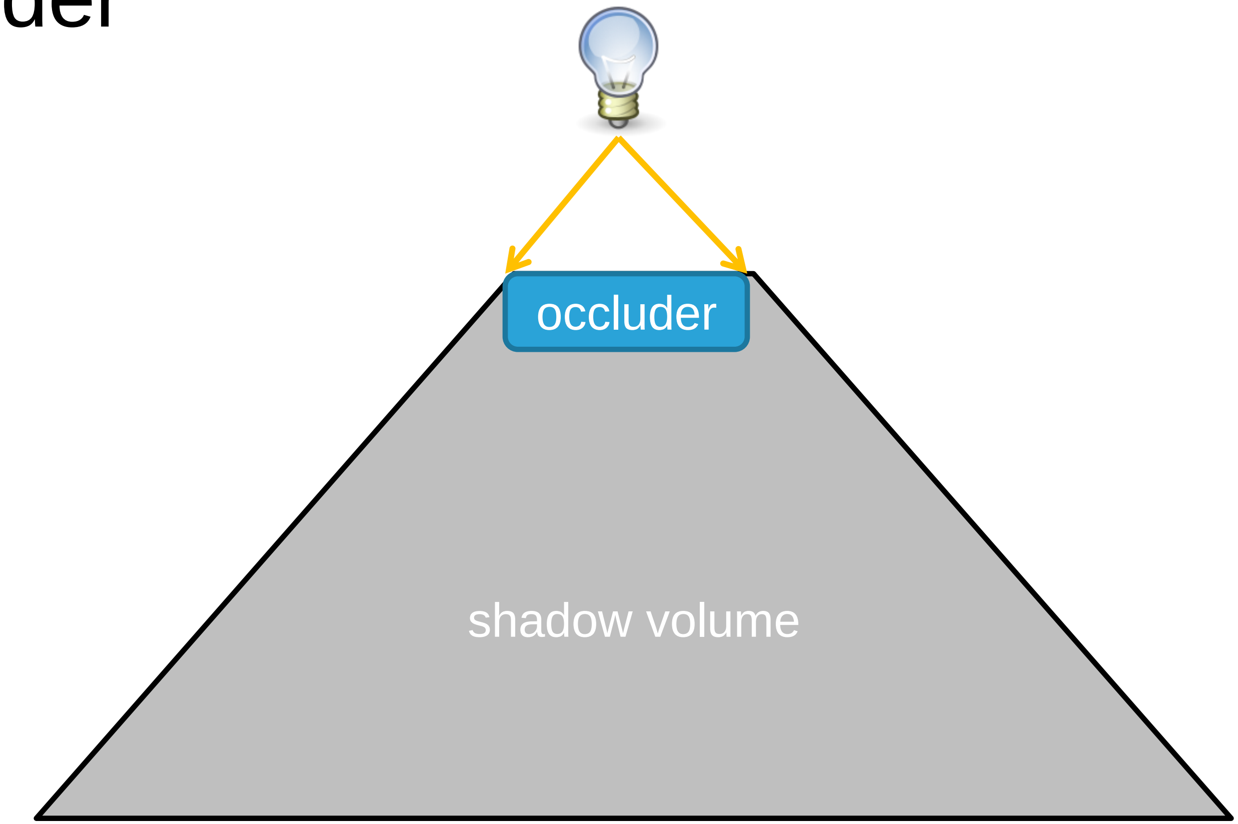


Shadow Volumes - History

- Technique has also been around for quite some time
  - Frank Crow, “Shadow Algorithms for Computer Graphics”, 1977

Shadow Volume - Terminology

- For given light source, shadow volume defines region of space that is in shadow of particular occluder


occluder
shadow volume

Stencil Buffer 101

- Stencil Buffer is useful GPU-feature for hardware accelerated implementation of the “is fragment in shadow?”-test
- Stencil Buffer almost always 8bit
  - Forms 32bit word together with 24bit z-buffer
- Supports basic tests and arithmetic operations
- Used to mask out complex shapes
  - Actually similar to depth buffer, but more flexible

Stencil Buffer 101

- Conditionally eliminate a pixel based on the outcome of comparison between reference val and current pixel‘s stencil val

```
glStencilFunc(
  GL_EQUAL,  // stencil comparison function
  0, // reference val
  ~0 // AND-mask
);

pixel[x][y] passes if((ref & mask) == (stencil[x][y] & mask))
```

Stencil Buffer 101

- Specify how to update stencil buffer based on several conditions

```
glStencilOpSeparate(
  GL_FRONT, //is front and/or back stencil state updated?
  GL_KEEP, //stencil test fails=> do not change stencil[x][y]
  GL_DECR, //if stencil passes but depth test fails=> stencil[x][y]--
  GL_INCR //if stencil passes, and depth passes=> stencil[x][y]++
);
```

Stencil Buffer 101

- Limited range of stencil values (only 8bit) can cause trouble
  - Slightly alleviated through wrap-around arithmethic

```
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_INCR_WRAP);
GL_INCR_WRAP ... Increment with wrap-around, i.e. 255++ => 0
```

  - Can still cause aliasing artifacts at multiples of 256; consider stencil value 0
    - 0 mod 256 = 0
    - 256 mod 256 = 0
    - k * 256 mod 256 = 0

Shadow Volume Overview

- Render scene with ambient and emissive lighting only
  - Also establishes z-buffer
- Determine shadow volume surface
  - Completely done in GS
- Render shadow volume surface
  - Update only stencil buffer
- Pixels outside shadow volume have stencil value zero
- Render scene again with diffuse and specular lighting
  - Additively blend with ambient lighting already in framebuffer
  - Rasterize only fragments having stencil value zero …
  - … and if depth(fragment)==zBuffer[x][y]

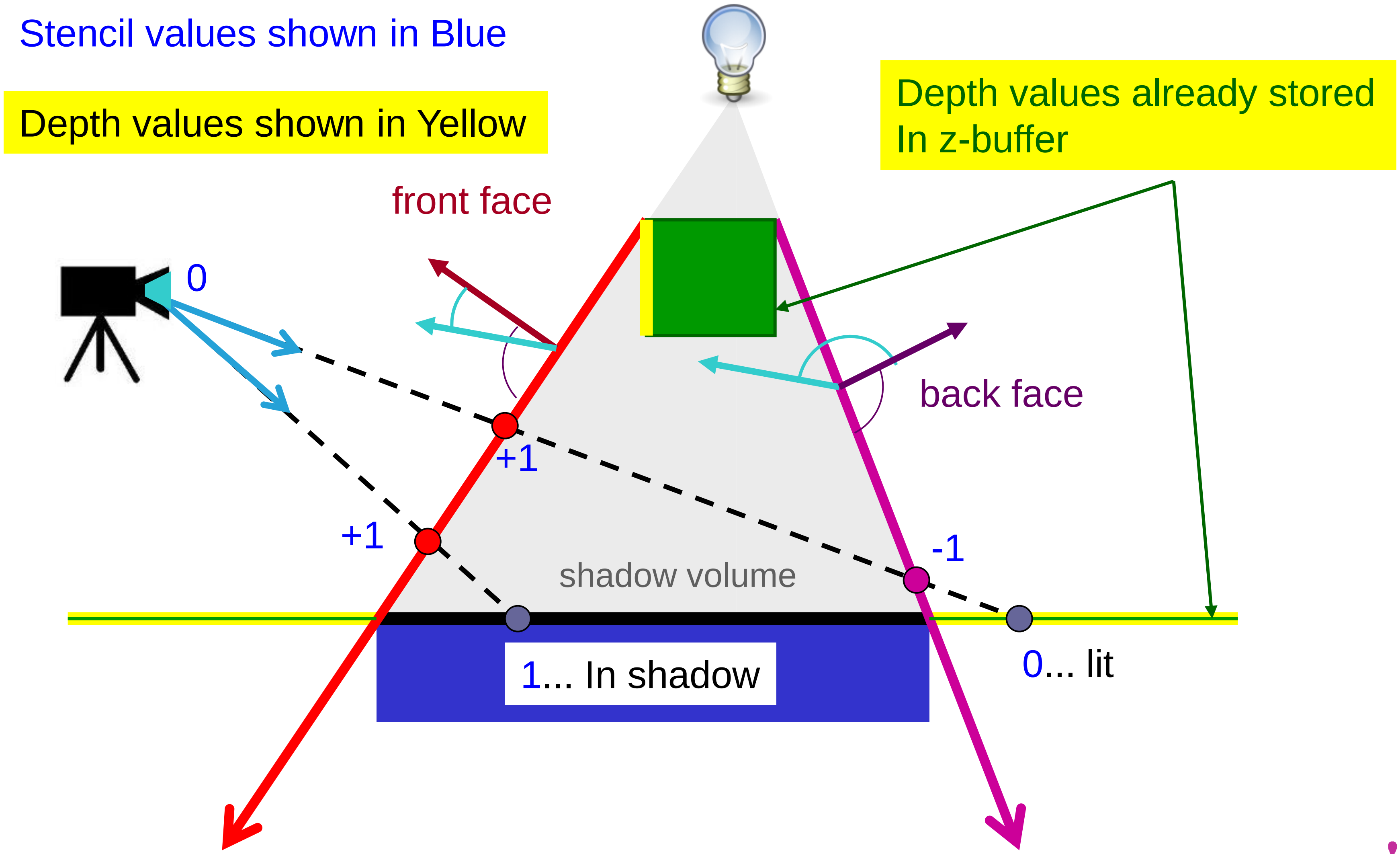
z-Pass vs. z-Fail

- Basic idea:
  - Start counting at 0
  - Increment counter at shadow volume entry points
  - Decrement counter at shadow volume exit points
  - If counter equals zero when geometry is hit, then we are not in the shadow volume (i.e. fragment is lit), otherwise we are in shadow

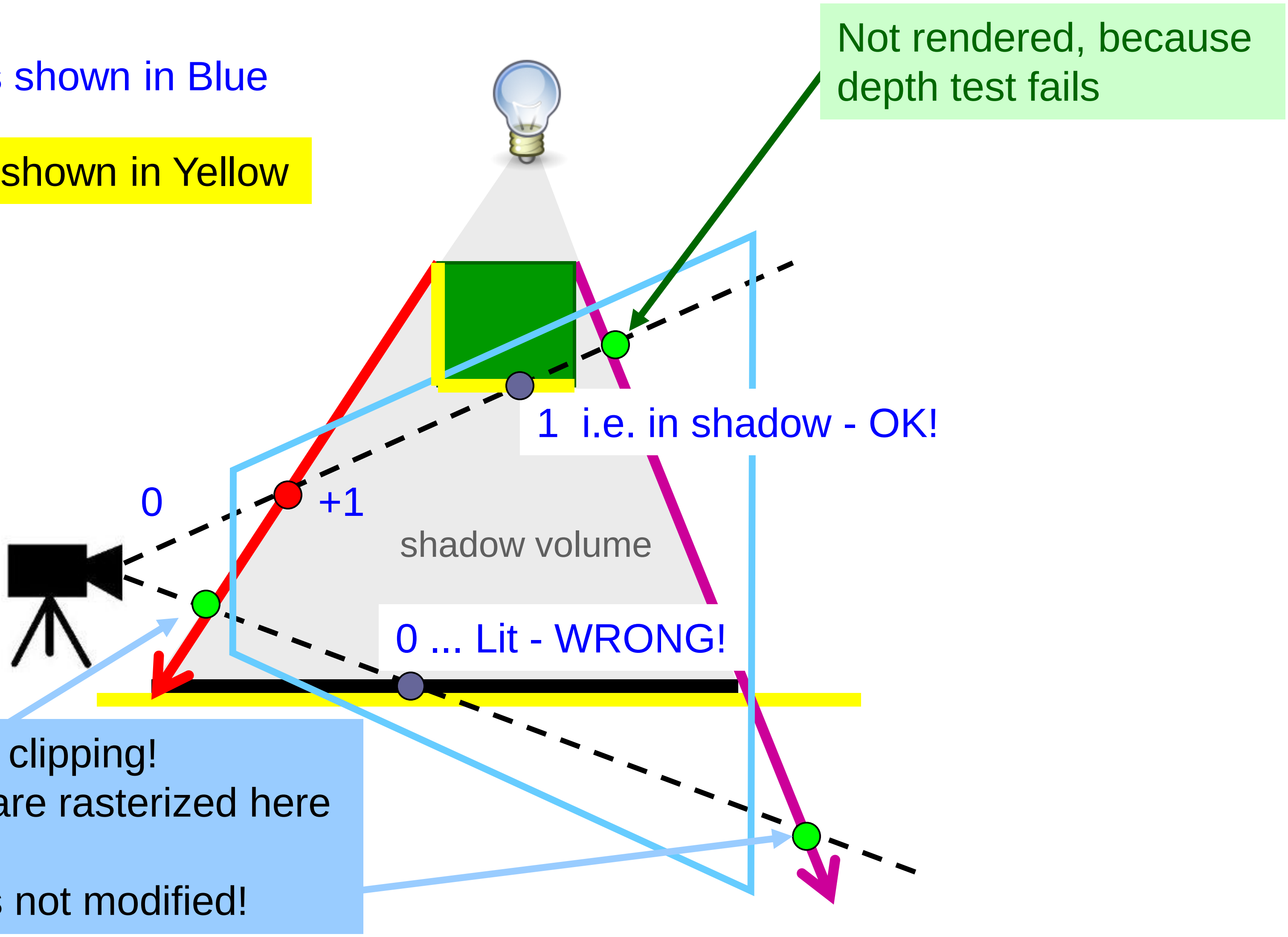
z-Pass vs. z-Fail

- Difference
  - z-Pass starts counting at camera along viewing ray until depth test fails (geometry is hit)
  - z-Fail starts counting at infinity and moves towards the eye until first time visible from the camera
    - Other way to look at this is starting at the eye and consider only points for which depth test fails, i.e. points which are further away from the eye than the first visible point

z-Pass


Stencil values shown in Blue
Depth values shown in Yellow
Depth values already stored
In z-buffer
front face
back face
0
+1
+1
-1
shadow volume
1... In shadow
0... lit

Why is z-Pass not Robust?



Stencil values shown in Blue

Depth values shown in Yellow

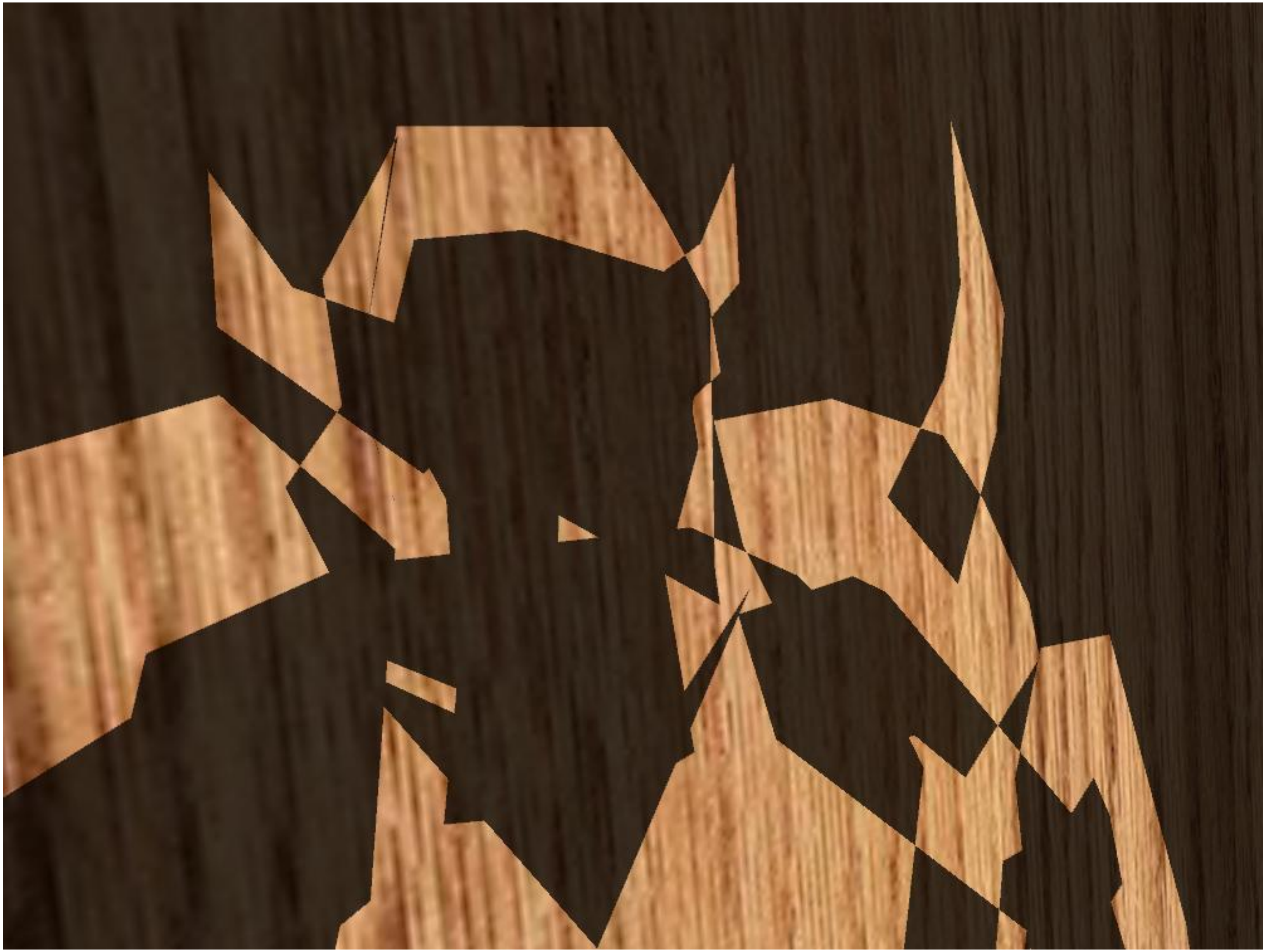
Not rendered, because depth test fails

1 i.e. in shadow - OK!

0 ... Lit - WRONG!

near-/far plane clipping!
No fragments are rasterized here
=>
stencil buffer is not modified!

z-Pass (Wrong)



z-Fail (Correct)

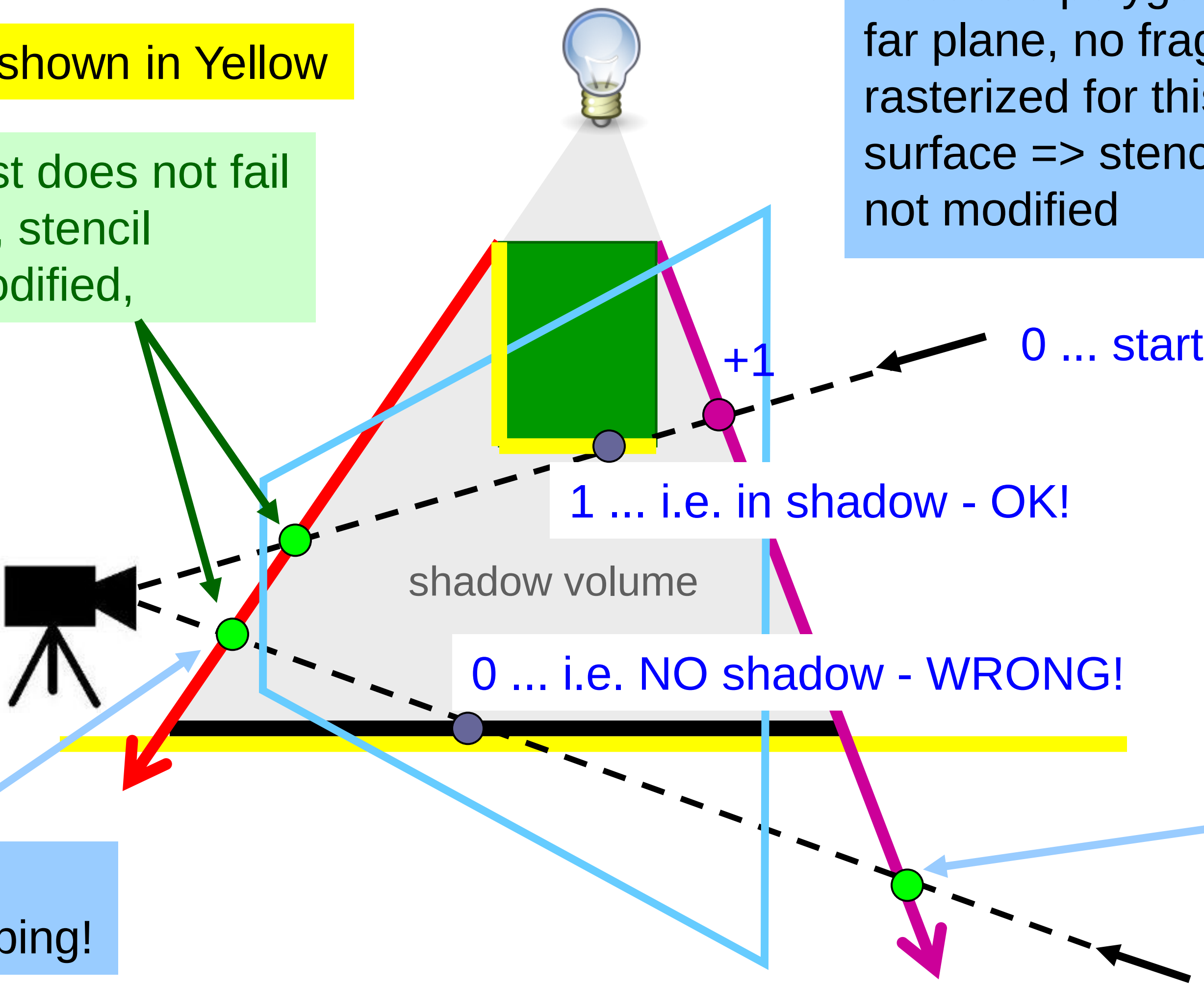
Is z-Fail Robust?

Stencil values shown in Blue

Depth values shown in Yellow

Since depth test does not fail (z test passes), stencil buffer is not modified,

Because polygon clipped by far plane, no fragment gets rasterized for this part of the surface => stencil buffer is not modified

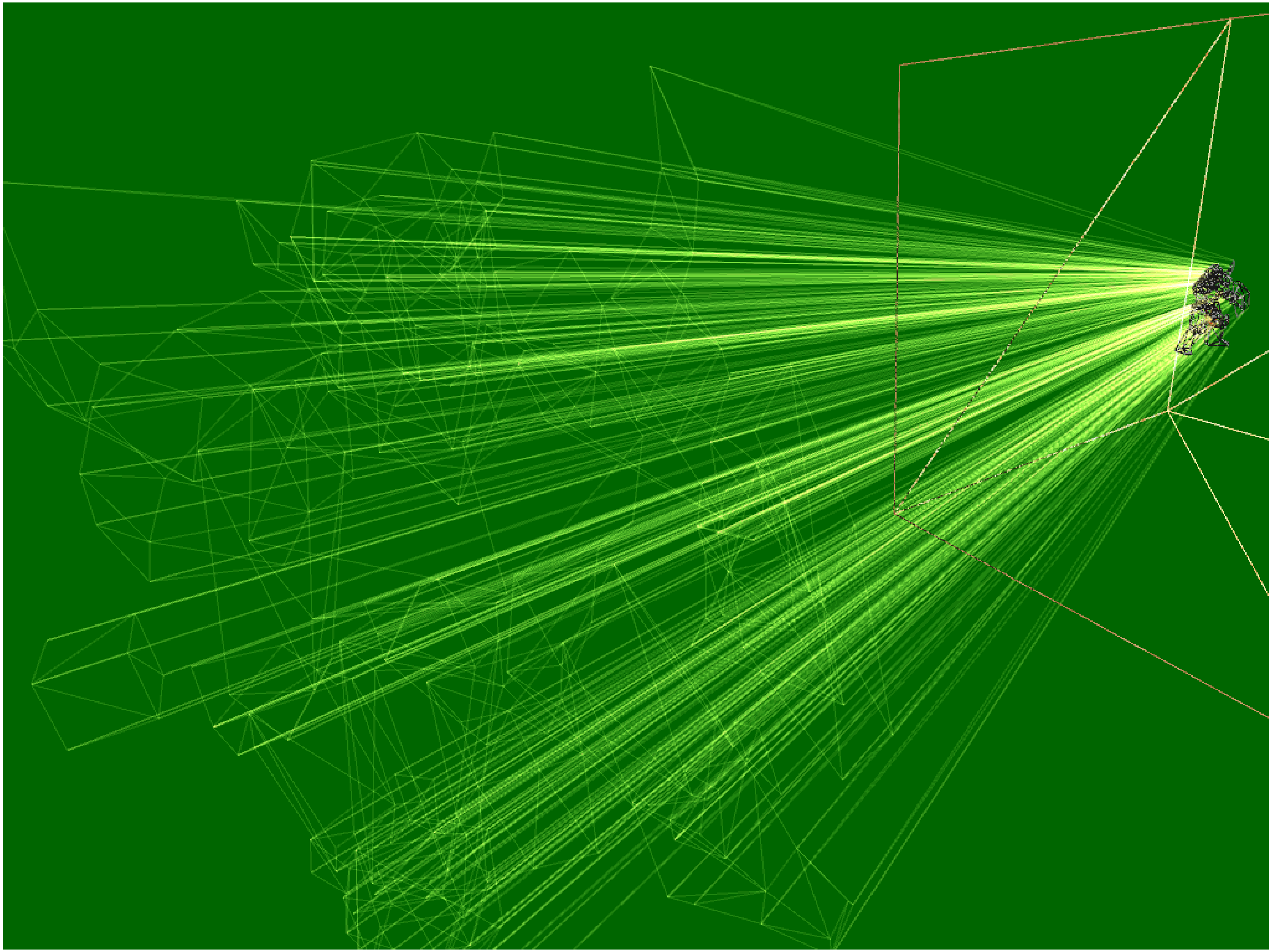

+1
0 ... start
1 ... i.e. in shadow - OK!
shadow volume
0 ... i.e. NO shadow - WRONG!
0 ... start

additionally:
near plane clipping!

z-Fail

- It seems we have only shifted the z-Pass problem of near plane clipping to the far plane
- At first sight this does not really help, right?
  - At least a lot easier to fix at the other end of the view-frustum ☺
  - Simply make sure to never clip shadow volume-mesh at the far plane
    - Depth clamp or infinite projection matrix
  - Close shadow volume-mesh from both sides
    - Light- and dark cap
  - Now counter does not get messed up

No Far Plane Clipping

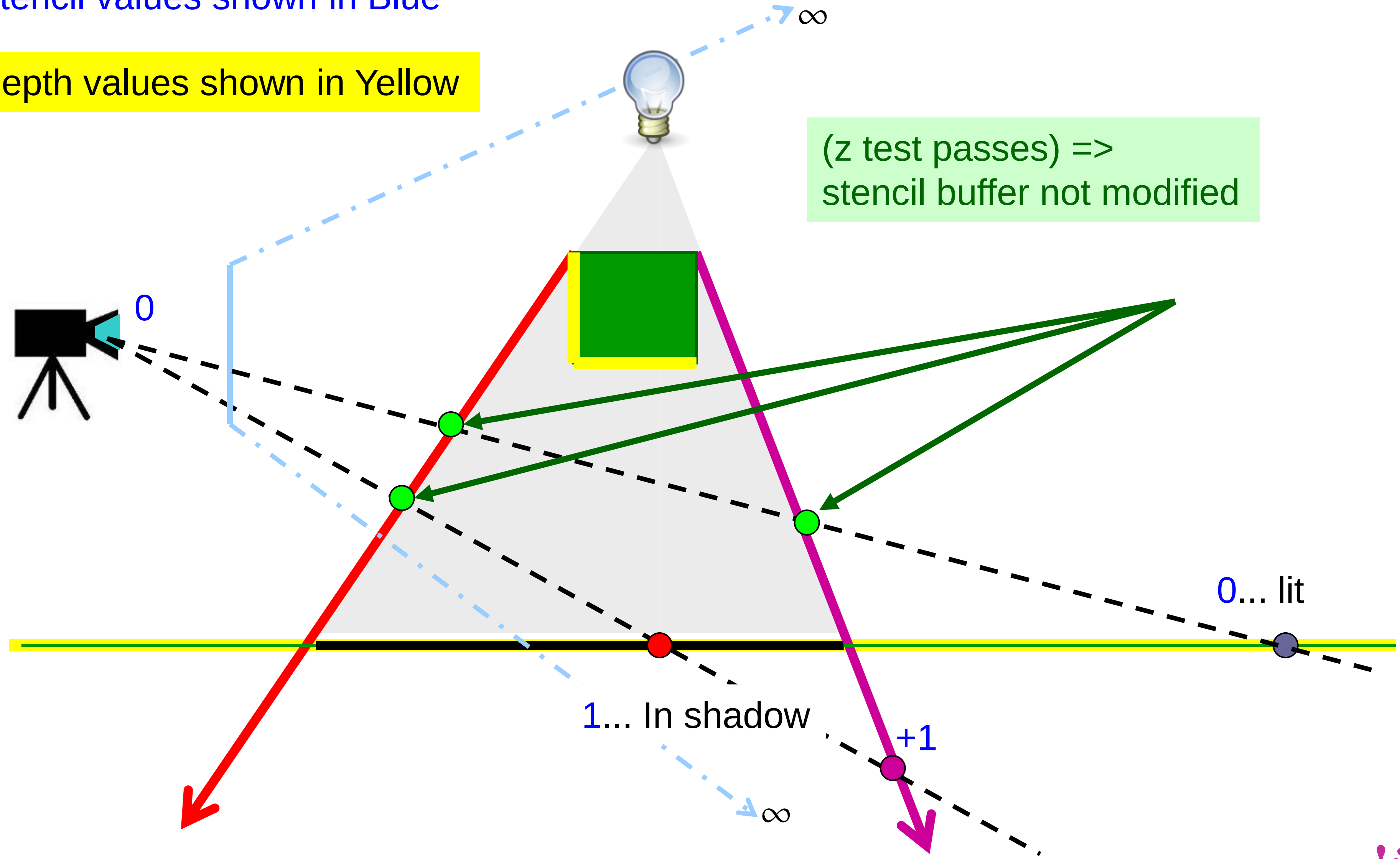


z-Fail

Stencil values shown in Blue

Depth values shown in Yellow

(z test passes) =>
stencil buffer not modified


∞
0
0... lit
1... In shadow
+1
∞

Each Frame

```
// make sure we clear buffers to desired values
glClearColor(0.0, 0.4, 0.0, 1.0);
glClearDepth(1.0); // clear to far plane distance in DC
glClearStencil(0);

// enable respective buffers for writing
// (in this case a "clear");
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
glStencilMask(~0);

// perform actual clear-buffers-operation
glClear(       GL_COLOR_BUFFER_BIT    |
               GL_DEPTH_BUFFER_BIT    |
               GL_STENCIL_BUFFER_BIT         );
```

Ambient Pass

```
glEnable(GL_CULL_FACE);

glDisable(GL_STENCIL_TEST);

glEnable(GL_DEPTH_TEST);

glDepthFunc(GL_LESS);

glDepthMask(GL_TRUE);

glDisable(GL_BLEND);

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

// render scene
```

Shadow Volume Pass

```
glEnable(GL_DEPTH_CLAMP);
glDisable(GL_CULL_FACE);
glEnable(GL_STENCIL_TEST);
if(zpass) {
  glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_INCR_WRAP);
  glStencilOpSeparate(GL_BACK, GL_KEEP, GL_KEEP, GL_DECR_WRAP);
}
else { //zfail
  glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_DECR_WRAP, GL_KEEP);
  glStencilOpSeparate(GL_BACK, GL_KEEP, GL_INCR_WRAP, GL_KEEP);
}
glStencilFunc(GL_ALWAYS, 0, ~0);
glStencilMask( ~0 );
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glDepthMask(GL_FALSE); // do not write to z-buffer
glDisable(GL_BLEND);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// render shadow volume polygons
glDisable(GL_DEPTH_CLAMP);
```

Diffuse+Specular Pass

```
glEnable(GL_CULL_FACE);
glEnable(GL_STENCIL_TEST);

//glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); //works well, but:
glStencilOp(GL_KEEP,GL_KEEP,GL_INCR);
// The INCR zpass stencil operation avoids double
// blending of lighting contributions in usually quite rare
// circumstance when two fragments alias to exact same pixel
// location and depth value

glStencilFunc(GL_EQUAL, 0, ~0);

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_EQUAL);

glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
```

Visualizing the Shadow Volume Surface

```
glEnable(GL_DEPTH_CLAMP);

glDisable(GL_CULL_FACE);

glEnable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);

//glDepthFunc(GL_LESS); //works well, but:
glDepthFunc(GL_LEQUAL); //works better for depth-clamp

//render surface transparently
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE); //additive alpha-blending
```

How to Determine Shadow Volume Polys

- Until now we looked on application-side (OpenGL state-settings) only
- What happens on the GPU?
  - Before looking at shaders a small, but important detail is still missing
  - In GS we need adjacency information for each triangle
  - Boils down to sending 6 vertices per triangle instead of only 3

- By simply passing 3 additional vertices per triangle we have access to the three neighbor triangles along the triangle-edges

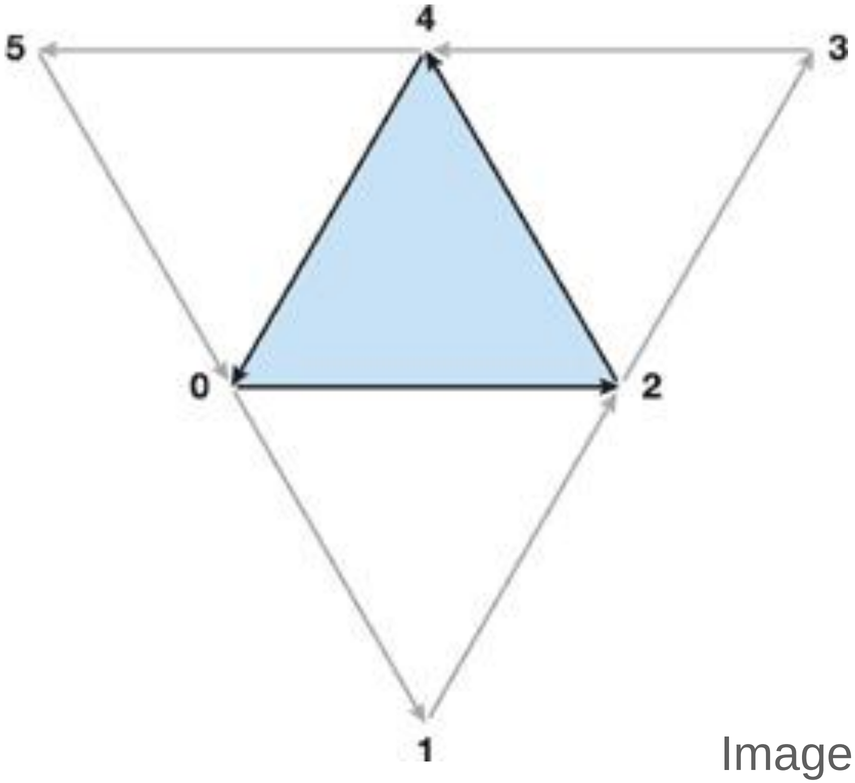


Image taken from [GPUGems3]

Adjacent triangles

- For every shadow caster store vertex data and
  - Create index for standard triangle rendering
    - 3 vertices make up a triangle
  - Create index for adjacent triangle rendering
    - 6 vertices make up a triangle
- Whenever rendering the shadow-volume we need the adjacency information
- For more info, see [Len1]

Tackling possible Artifacts

- In final light-blend pass we rely on depth test for equal z-value
    - Problem when using multiple passes due to numerical errors
    - Make sure transformations “match exactly“
    - So employ same vertex shader for standard- and for shadow volume rendering
    - Declare position as **invariant**

(Shared) SV-Vertex Shader

```
#version 330 core

uniform mat4 PV_mat; // (projection * view) matrix
uniform mat4 M_mat; // model matrix

in vec3 attr_vertex; // object space vertex position
invariant out vec4 WS_pos; // to be passed on to GS
// and possibly other uniforms and varyings

void main(void) {
  // ...
  WS_pos = M_mat * vec4(attr_vertex, 1.0);

  vec4 CS_pos = PV_mat * WS_pos;

  // transform to CS as usual,
  // so VS still works for standard rendering
  gl_Position = CS_pos;
}
```

(Shared) SV-Geometry Shader (1/8)

```
#version 330 core
//our primitive is made up of 6 vertices
layout(triangles_adjacency) in;
layout(triangle_strip) out; //write out triangle strips

//(3 + 3 for the two caps plus 4 x 3 for the sides)
layout(max_vertices=18) out;

uniform mat4 PV_mat; // (projection * view) matrix

uniform vec4 l_pos;  // Light position (world space)
uniform int zpass;  // Is it safe to do z-pass?

// passed from VS
// array[6] because our primitive is made up of 6 vertices
invariant in vec4 WS_pos[6];

// and possibly other uniforms and varyings
```

(Shared) SV-Geometry Shader (2/8)

```
void main(void)
{

   vec3 ns[3];  // Normals
   vec3 d[3];  // Directions toward light
   vec4 v[4];  // Temporary vertices

   // Triangle oriented toward light source
   vec4 or_pos[3];
   or_pos[0] = WS_pos[0];
   or_pos[1] = WS_pos[2];
   or_pos[2] = WS_pos[4];
```

(Shared) SV-Geometry Shader (3/8)

```
// Compute normal at each vertex.
ns[0] = cross(
   WS_pos[2].xyz - WS_pos[0].xyz,
   WS_pos[4].xyz - WS_pos[0].xyz );
ns[1] = cross(
   WS_pos[4].xyz - WS_pos[2].xyz,
   WS_pos[0].xyz - WS_pos[2].xyz );
ns[2] = cross(
   WS_pos[0].xyz - WS_pos[4].xyz,
   WS_pos[2].xyz - WS_pos[4].xyz );

// Compute direction from vertices to light.
d[0] = l_pos.xyz-l_pos.w*WS_pos[0].xyz;
d[1] = l_pos.xyz-l_pos.w*WS_pos[2].xyz;
d[2] = l_pos.xyz-l_pos.w*WS_pos[4].xyz;
```

(Shared) SV-Geometry Shader (4/8)

```
// Check if the main triangle faces the light.
if ( !(dot(ns[0],d[0])>0 || dot(ns[1],d[1])>0 ||
       dot(ns[2],d[2])>0) ) {
  return;  // Not facing the light => irrelevant for SV
}
// when we get here, we know current triangle is facing the light
const bool faces_light=true;
```

(Shared) SV-Geometry Shader (5/8)

```
// Render caps - only needed for z-fail.
  if ( zpass == 0 ) {

    // Near cap - simply render triangle
    gl_Position = PV_mat*or_pos[0];  EmitVertex();
    gl_Position = PV_mat*or_pos[1];  EmitVertex();
    gl_Position = PV_mat*or_pos[2];  EmitVertex();
    EndPrimitive();

    // Far cap - extrude positions to infinity (w=0)
    // note the different triangle-winding order (0-1-2 => 0-2-1)
    v[0] = vec4(l_pos.w*or_pos[0].xyz-l_pos.xyz,0);
    v[1] = vec4(l_pos.w*or_pos[2].xyz-l_pos.xyz,0);
    v[2] = vec4(l_pos.w*or_pos[1].xyz-l_pos.xyz,0);

    gl_Position = PV_mat*v[0];  EmitVertex();
    gl_Position = PV_mat*v[1];  EmitVertex();
    gl_Position = PV_mat*v[2];  EmitVertex();
    EndPrimitive();
 }
```

(Shared) SV-Geometry Shader (6/8)

```
// Loop over all edges and extrude if needed.
  for ( int i=0; i<3; i++ ) {
    // Compute indices of neighbor triangle.
    int v0 = i*2;
    int nb = (i*2+1);
    int v1 = (i*2+2) % 6;

    // Compute normals at vertices, the *exact*
    // same way as done above!
    ns[0] = cross(
      WS_pos[nb].xyz-WS_pos[v0].xyz,
      WS_pos[v1].xyz-WS_pos[v0].xyz);
    ns[1] = cross(
      WS_pos[v1].xyz-WS_pos[nb].xyz,
      WS_pos[v0].xyz-WS_pos[nb].xyz);
    ns[2] = cross(
      WS_pos[v0].xyz-WS_pos[v1].xyz,
      WS_pos[nb].xyz-WS_pos[v1].xyz);
```

(Shared) SV-Geometry Shader (7/8)

```
// Compute direction to light, again as above.
     d[0] =l_pos.xyz-l_pos.w*WS_pos[v0].xyz;
     d[1] =l_pos.xyz-l_pos.w*WS_pos[nb].xyz;
     d[2] =l_pos.xyz-l_pos.w*WS_pos[v1].xyz;
```

(Shared) SV-Geometry Shader (8/8)

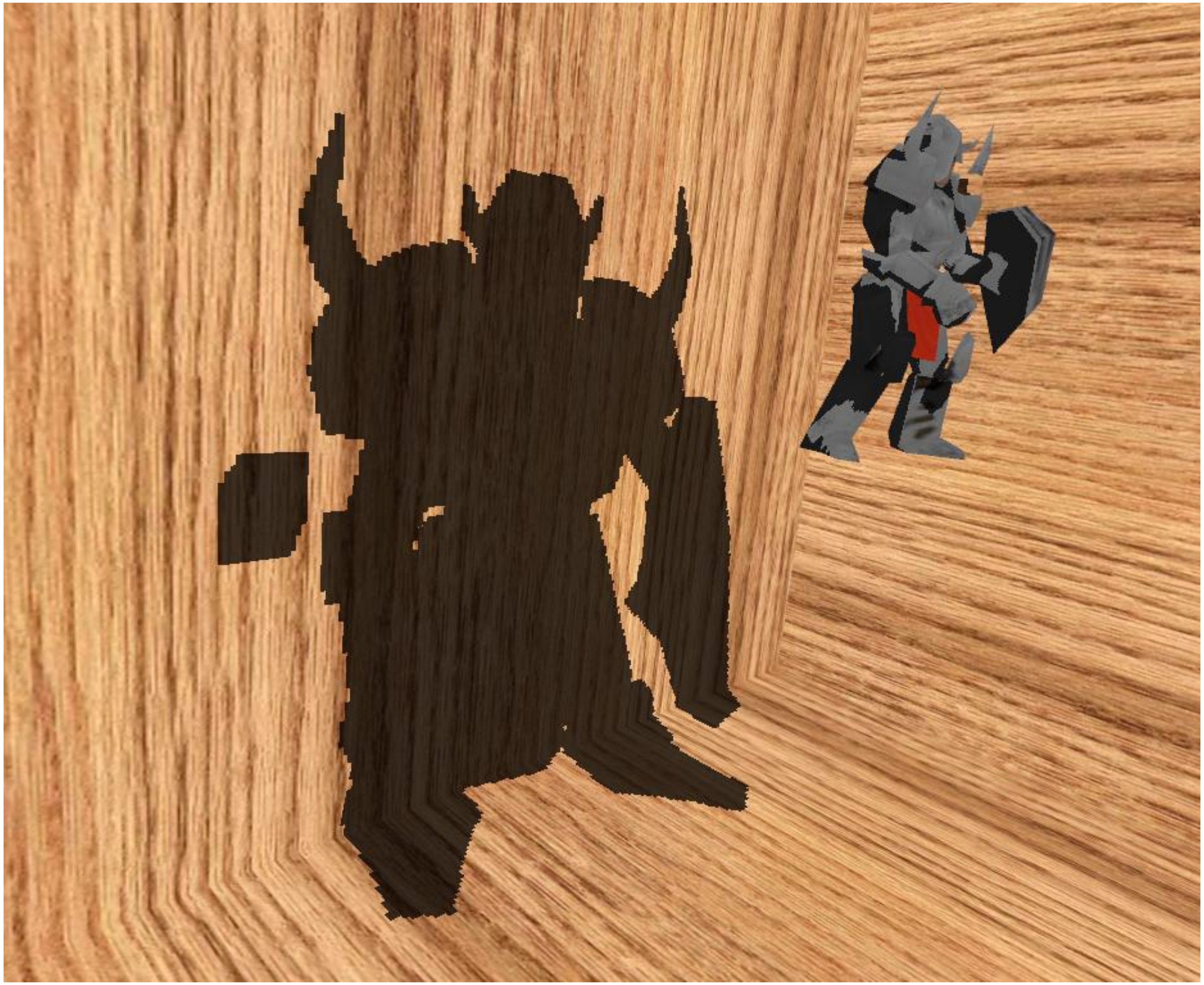
```
   // Extrude the edge if it does not have a
   // neighbor, or if it's a possible silhouette.
   if ( WS_pos[nb].w<0.001 || (faces_light!=(dot(ns[0],d[0])>0 ||
                                             dot(ns[1],d[1])>0 ||
                                             dot(ns[2],d[2])>0) )) {
       // Make sure sides are oriented correctly.
       int i0 = faces_light ? v0 : v1;
       int i1 = faces_light ? v1 : v0;
       v[0] = WS_pos[i0];
       v[1] = vec4(l_pos.w*WS_pos[i0].xyz - l_pos.xyz, 0);
       v[2] = WS_pos[i1];
       v[3] = vec4(l_pos.w*WS_pos[i1].xyz - l_pos.xyz, 0);
       // Emit a quad as a triangle strip.
       gl_Position = PV_mat*v[0];  EmitVertex();
       gl_Position = PV_mat*v[1];  EmitVertex();
       gl_Position = PV_mat*v[2];  EmitVertex();
       gl_Position = PV_mat*v[3];  EmitVertex();
       EndPrimitive();
     }
  }
}
```

(Shared) SV-Fragment Shader

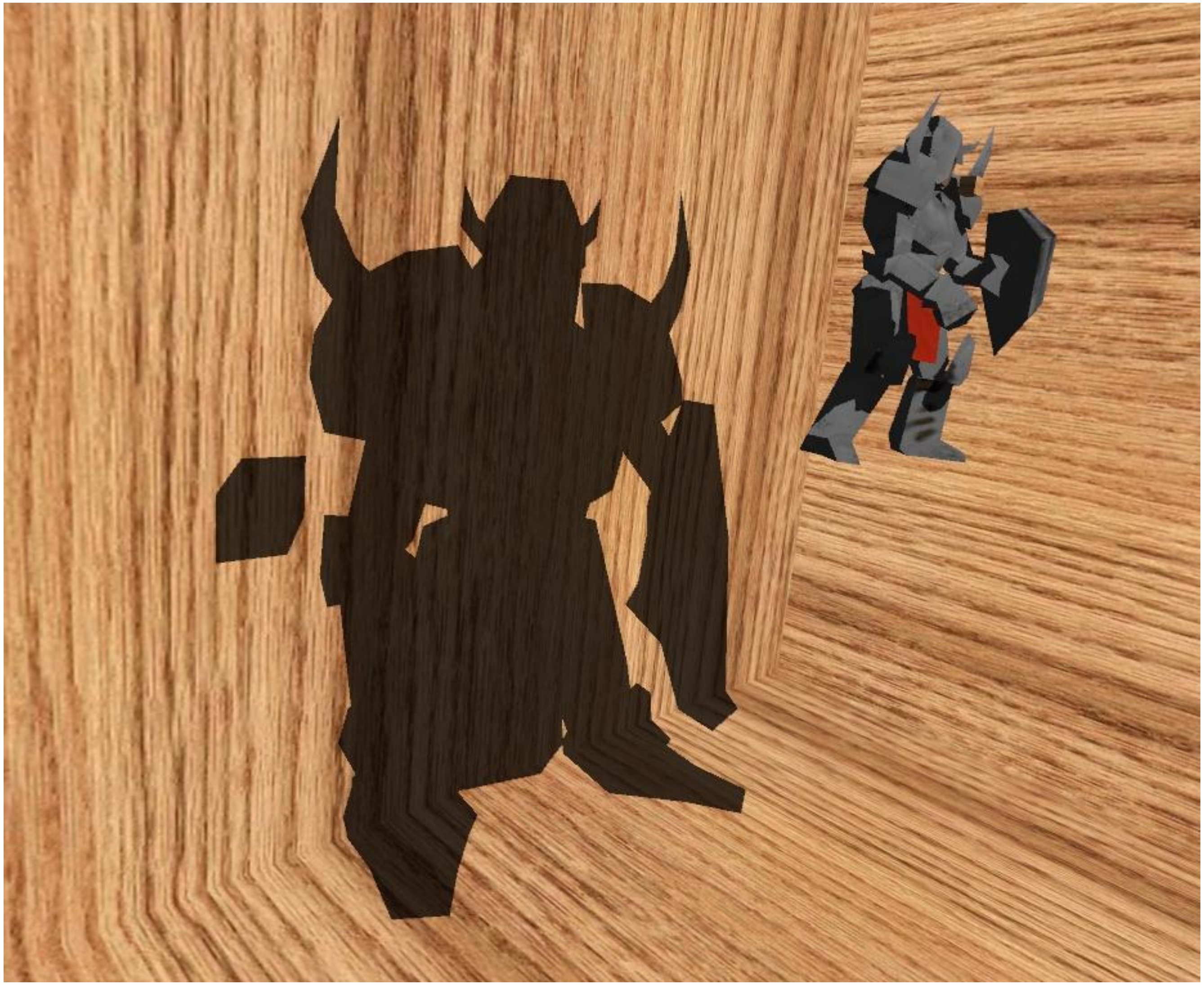
```
#version 330 core

out vec4 frag_data_0;

void main(void)
{
  // color value actually only used when visualizing
  // shadow volume mesh
  // important thing happens implicitly (compare to depth buffer!):
  // stencil buffer is updated according to previous
  // state-configuration from the app
  frag_data_0 = vec4(0.25, 0.25, 0.125, 0.25);
}
```

- Light sources must be ideal points. Homogeneous light
- positions ($w \geq 0$) allow both positional and directional lights.

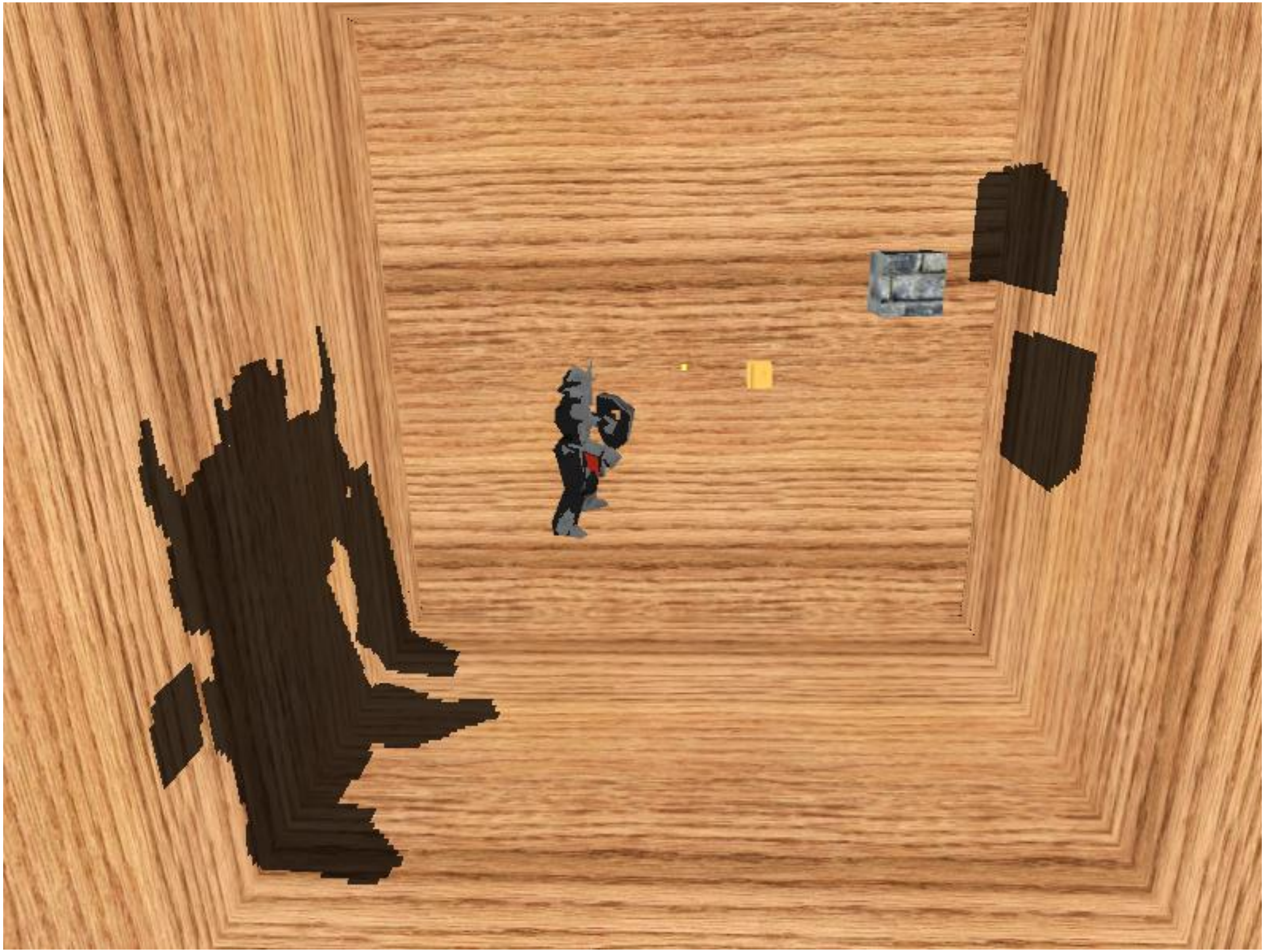
Comparison - CMSM



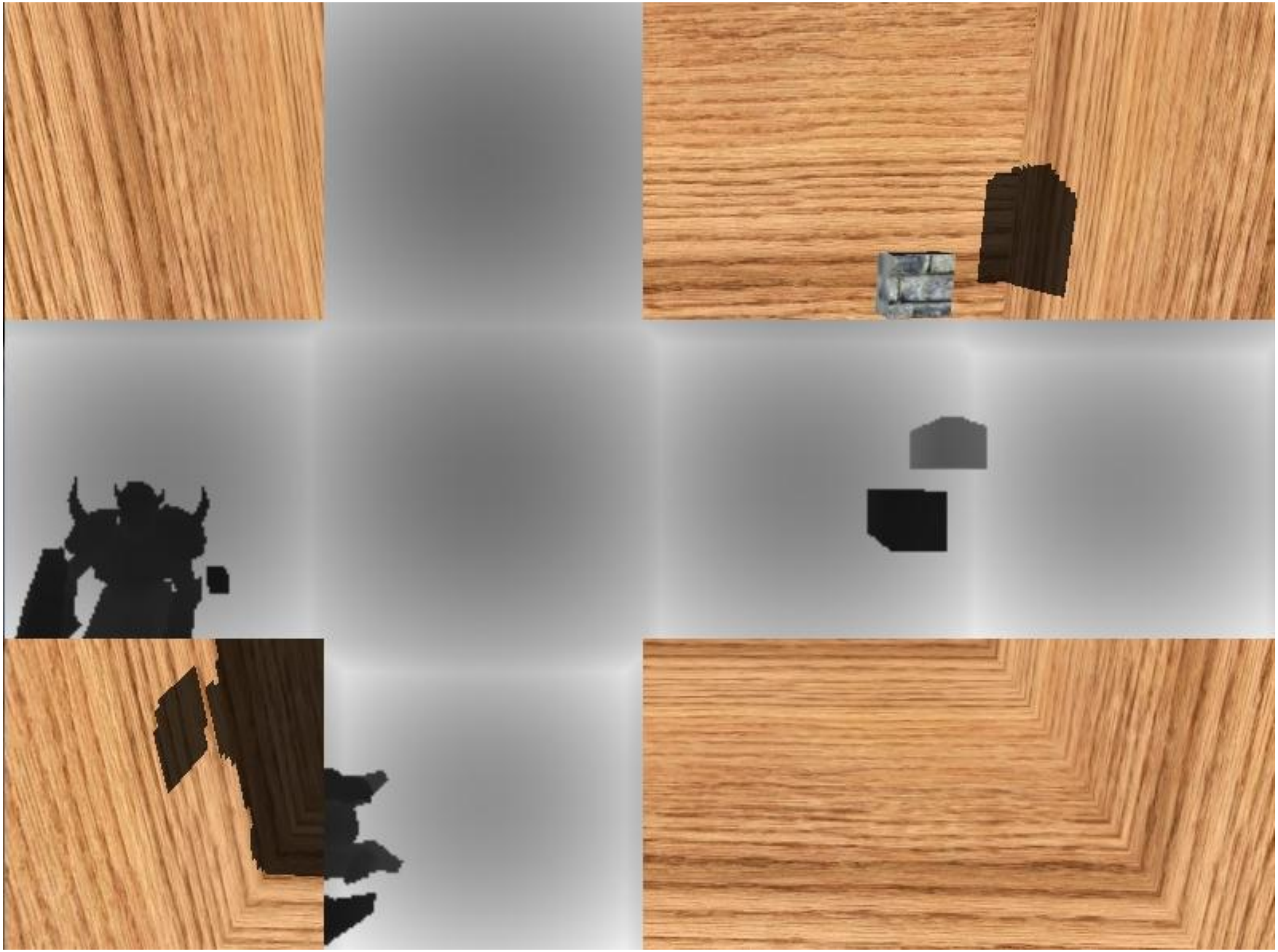
Comparison - SV



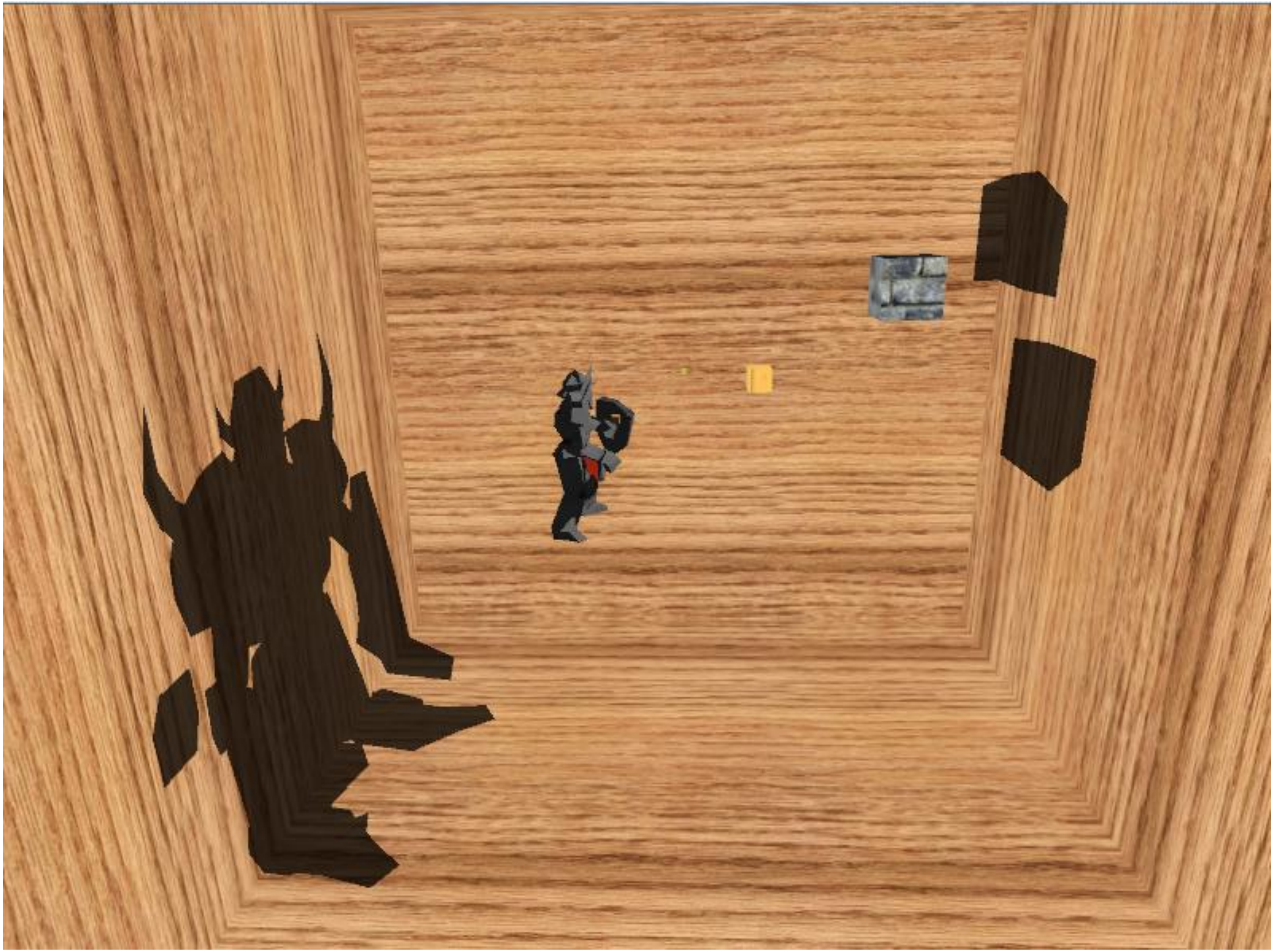
CMSM



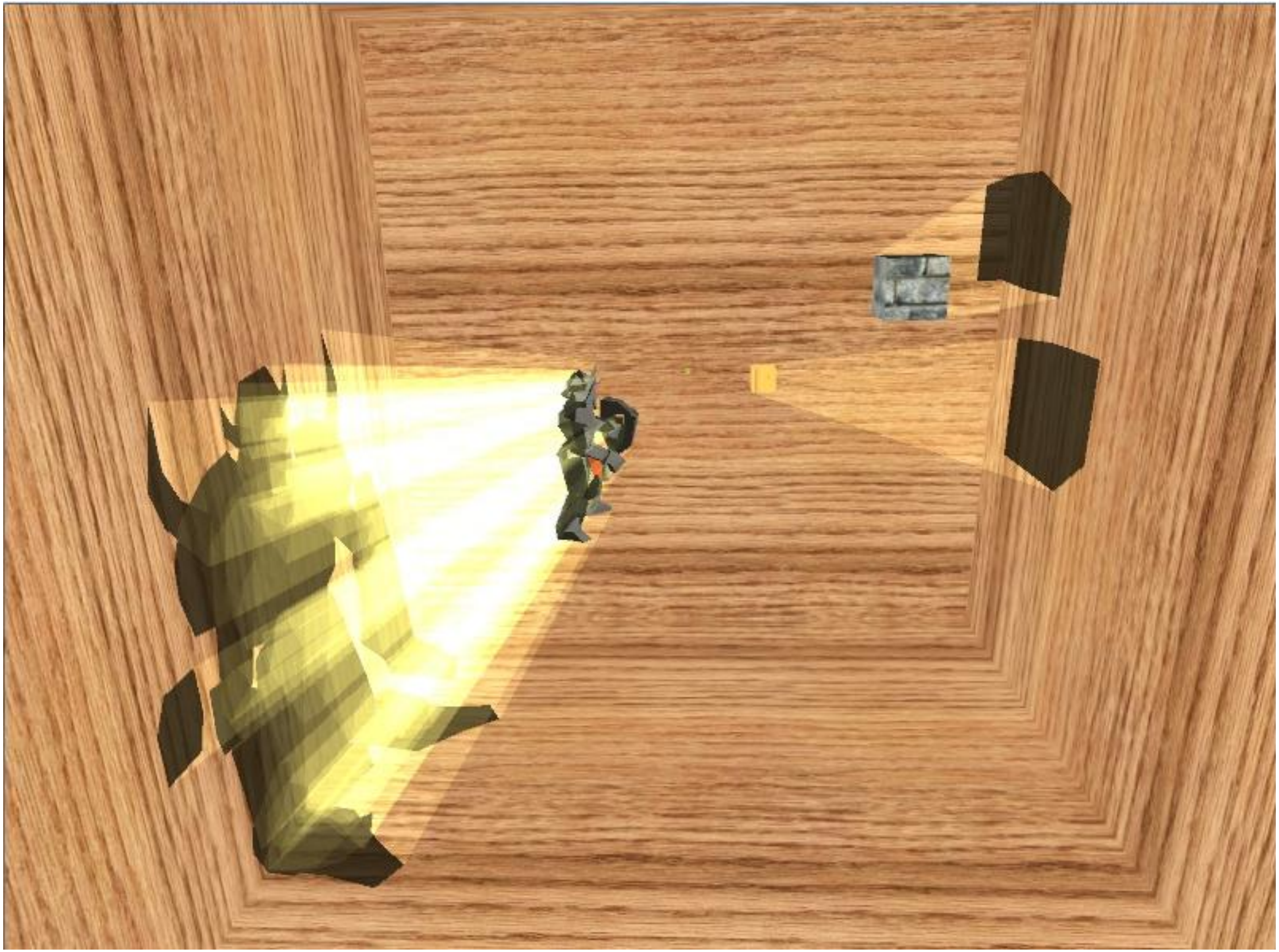
CMSM



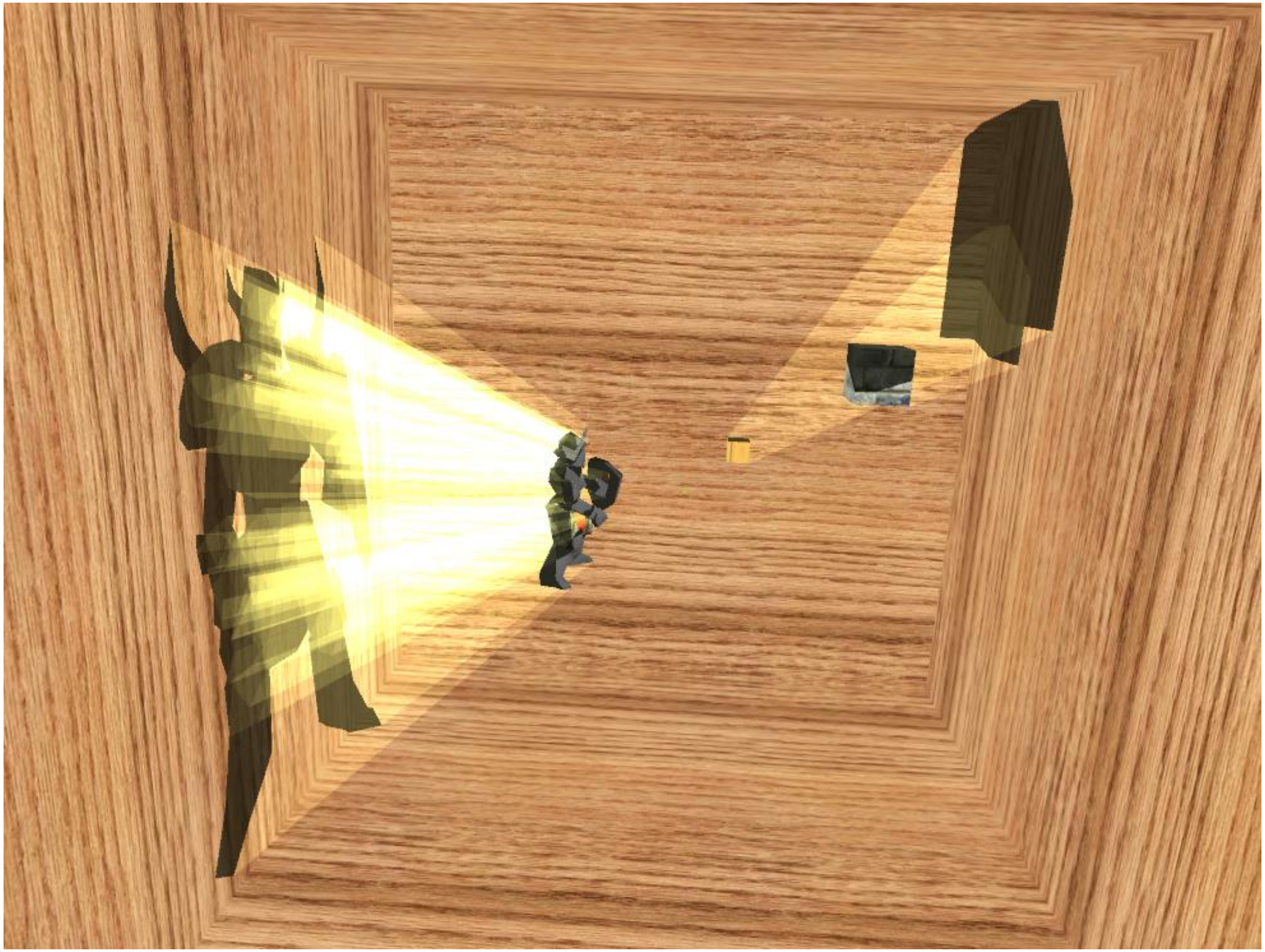
SV



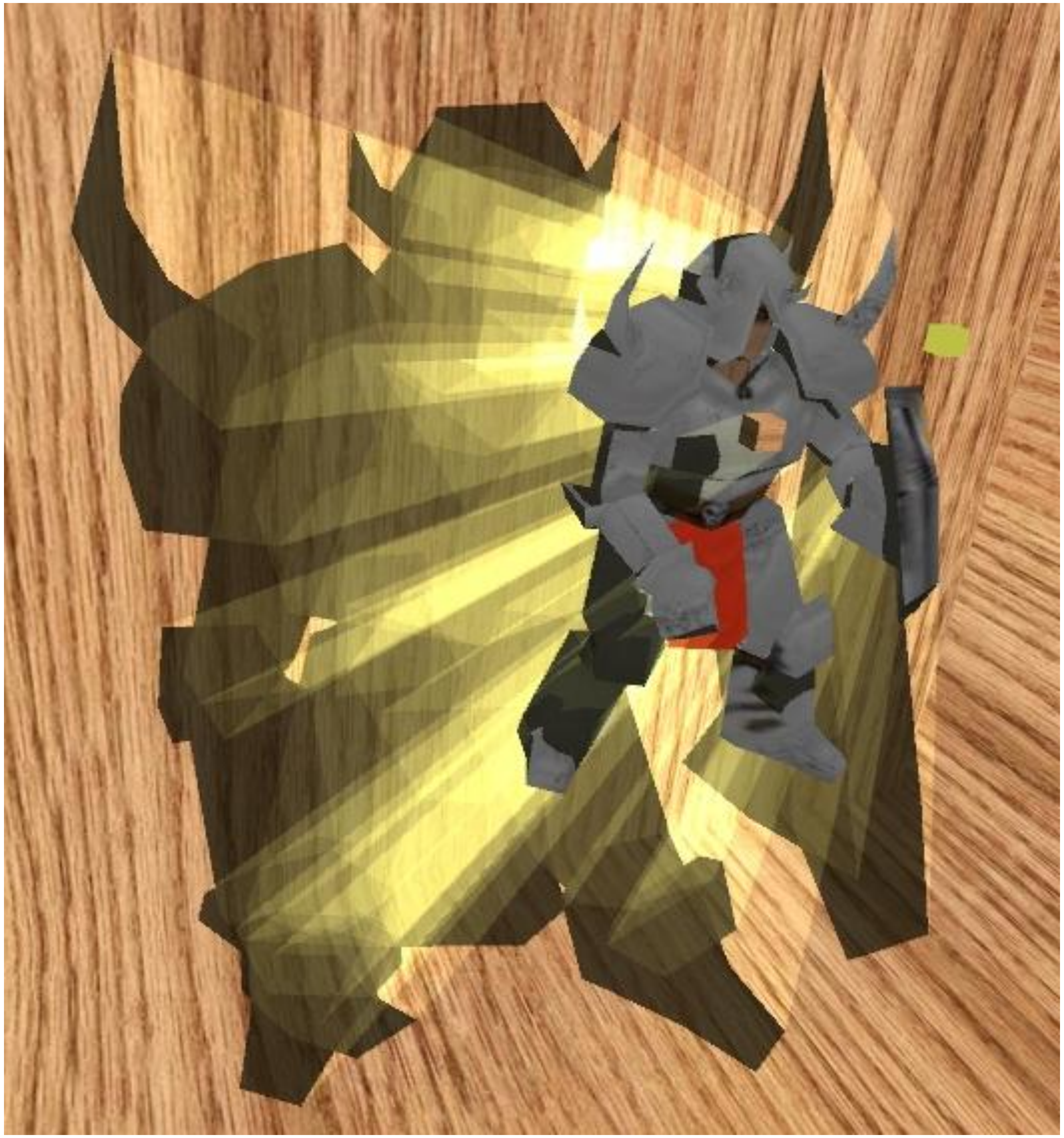
SV



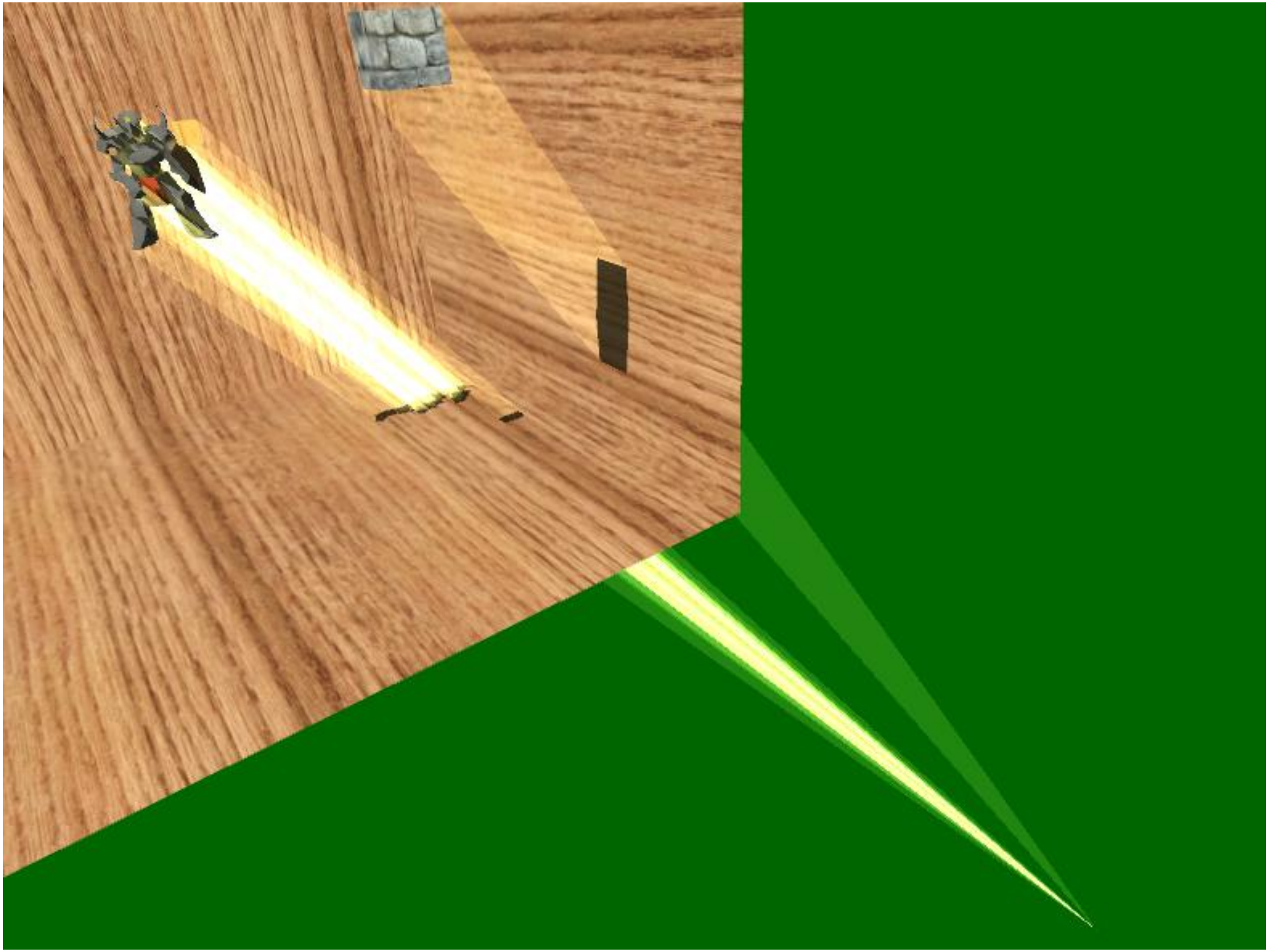
SV



Point Light (light_pos.w=1)



Directional Light (light_pos.w=0)



References

- http://http.developer.nvidia.com/GPUGems/gpugems_ch12.html
- http://http.developer.nvidia.com/GPUGems/gpugems_ch09.html
- [Len1] http://www.terathon.com/code/edges.html
- [GPUGems3] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch11.html

- Special thanks to Eric Lengyel, Mark Kilgard and Cass Everitt!